下载APP

28 | 增加更丰富的类型第3步：支持数组

2021-10-18 宫文学

《手把手带你写一门编程语言》 课程介绍 >

讲述：宫文学
时长 19:37 大小 17.97M

你好，我是宫文学。

前面我们已经给我们的语言，增加了两种数据类型：浮点数和字符串。那么，今天这一节课，我们再继续增加一种典型的数据类型：数组。

数组也是计算机语言中最重要的基础类型之一。像 C、Java 和 JavaScript 等各种语言，都提供了对数组的原生支持。

数组跟我们之前已经实现过的 number、string 类型的数据类型相比，有明显的差异。所以在这里，你也能学到一些新的知识点，包括，如何对数组做类型处理、如何设计数组的内存布局，如何正确访问数组元素，等等。

在完成这节课的任务后，我们的语言将支持在数组中保存 number 和 string 类型的数据，甚至还可以支持多维数组。是不是感觉很强大呢？那就赶紧动手试一试吧！

那么这实现的第一步，我们需要修改编译器前端的代码，来支持与数组处理有关的语法。

修改编译器前端

在编译器前端，我们首先要增加与数组有关的语法规则。要增加哪些语法规则呢？我们来看看一个最常见的使用数组的例子中，都会涉及哪些语法特性。

复制代码

```
let names:string[] = ["richard", "sam", "john"];
let ages:number[] = [8, 18, 28];
let a2:number[][] = [[1,2,3],[4,5]];
for (let i = 0; i< names.length; i++){
  println(names[i]);
}
```

在这个例子中，我们首先声明了一个字符串类型的数组，然后用一个数组字面量来初始化它。你还可以用同样的方法，声明并初始化一个 number 数组。最后，我们用 names[i]这样的表达式来访问数组元素。

在这个例子中，你会发现三个与数组有关的语法现象，分别是**数组类型**、**数组字面量**和**下标表达式**。

首先是数组类型。在声明变量的时候，我们可以用 string[]、number[]来表示一个数组类型。这个数组类型是一个基础类型再加上一对方括号[]。在这里，我们甚至还声明了一个二维数组。

所以，我们还需要扩展与类型有关的语法规则。你看看下面的语法规则，**你可以这样表示数组类型：“primaryType ‘[’ ‘]’ ”。而 primaryType 本身也可以是一个数组类型，这样就能表达多维数组了，比如 a[][]。**

复制代码

```
primaryType : predefinedType | literal | typeReference | '(' type_ ')' | prima
```

不过，这是一个左递归的文法，就会遇到我们之前学过的左递归问题。我们可以改写一下，变成下面的文法：

```
1 primaryType : primaryTypeLeft ('[' ']')* ;
2 primaryTypeLeft : predefinedType | literal | typeReference | '(' type_ ')' | p
```

这样的话，我们每次解析完毕一个 primaryTypeLeft 以后，再看看后面有没有跟着一对方括号就行了。如果出现多对方括号，就表示这是一个多维数组。

第二个语法现象是数组字面量，比如 string 字面量[“richard”，“sam”，“john”]和 number 字面量[8, 18, 28]。

这里，我们也总结出了数组字面量的语法规则：**每个数组字面量都是以方括号括起来的，方括号里面可以有一组元素，这些元素可以是一个表达式，或者是一个标识符，中间以逗号分割。**

```
1 primary:  literal | functionCall | '(' expression ')' | typeOfExp | arrayLiter
2 arrayLiteral : ('[' elementList? ']');
3 elementList : arrayElement (','+ arrayElement)* ;
4 arrayElement : expression ','? ;
```

你要注意最后一条语法规则“arrayElement : expression ‘,’ ? ;”。这条语法规则的意思是，数组中的最后一个元素，后面仍然可以跟一个逗号。比如上面的 number 数组等价于[8, 18, 28,]。

第三个语法现象是下标表达式。我们可以用 names[i]来引用数组的第 i 个元素。

所以，我们要扩展表达式的语法规则，**让基础式后面可以跟上一对方括号，变成一个下标表达式**。修改后的语法规则如下。这个文法的书写方式也同样避免了左递归。

```
1 expression: 'typeof' expression | assignment ('[' expression ']')* ;
```

上面就是目前我们需要扩展的语法规则了。为了支持新的语法规则，我们要创建几个新的 AST 节点对象：一个是 ArrayPrimType，代表数组类型节点；第二个是 ArrayLiteral，代表数组字面量；还有一个是 IndexedExp，表示像 a[2]这样的带有 index 的下标表达式。

在这里，你要注意，凡是表达式后面跟一个[index]，都表示访问该表达式的第 n 个元素。这个表达式不一定是个数组，只要是能够用下标访问的对象就行。比如，你可以像下面的程序那样，用下标访问一个字符串中的字符。

复制代码

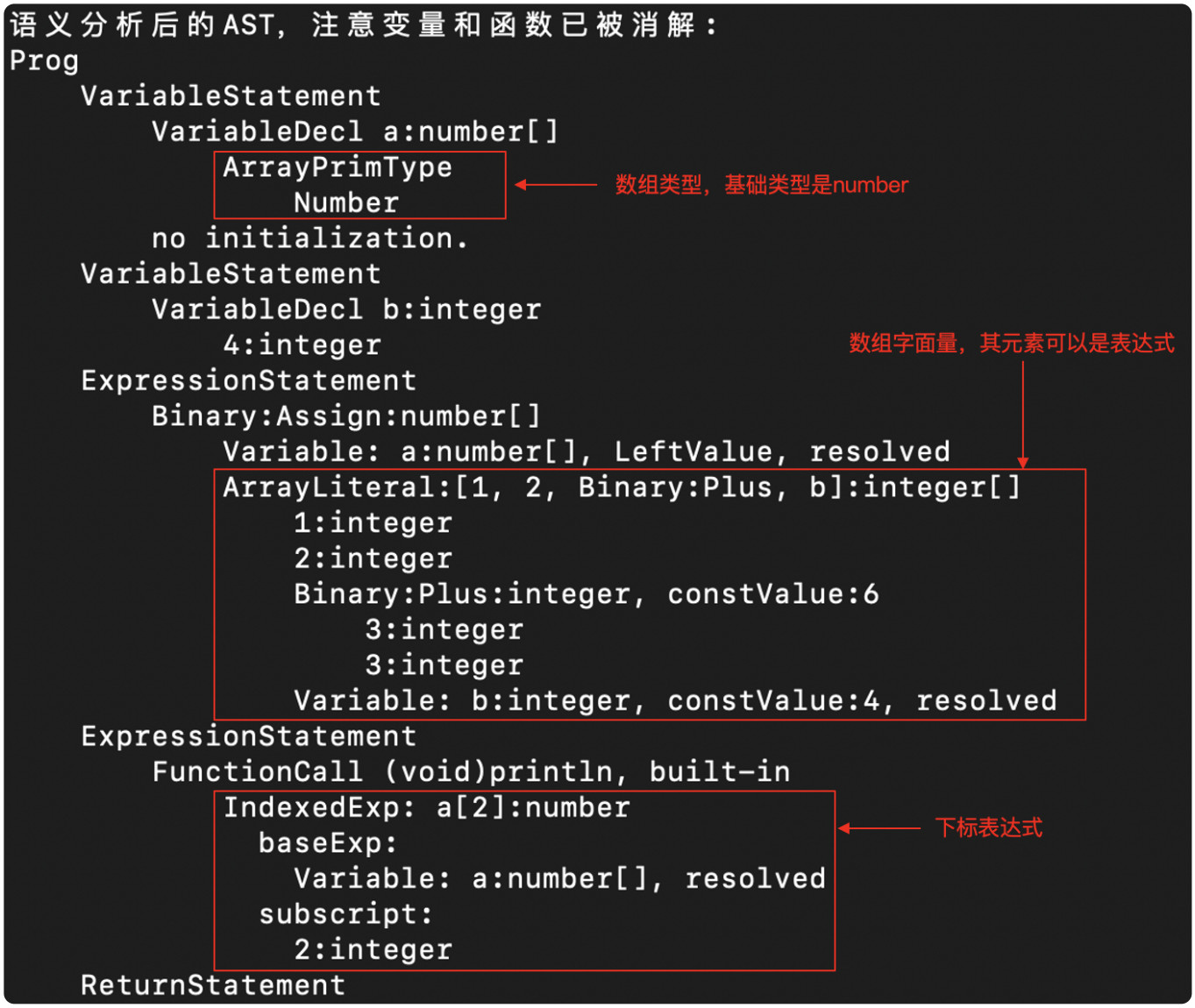
```
let name = "Richard";
name[1];   //返回'i'
```

在补充了相应的 AST 节点以后，我们就可以根据语法规则修改语法解析程序了。你可以试着运行下面这个的测试程序，用“node play example.ts -v”命令，带上参数 -v，打印出 AST 来看看。

复制代码

```
let a: number[];    //数组类型
let b = 4;
a = [1, 2, 3+3, b]; //数组字面量，其中的元素可以是表达式
println(a[2]);      //下标表达式
```

在终端输出的 AST 如下图所示，我在里面标注了上面讨论到的几种 AST 节点。



你还可以看看语法分析程序对多维数组的支持情况，比如下面的示例程序：

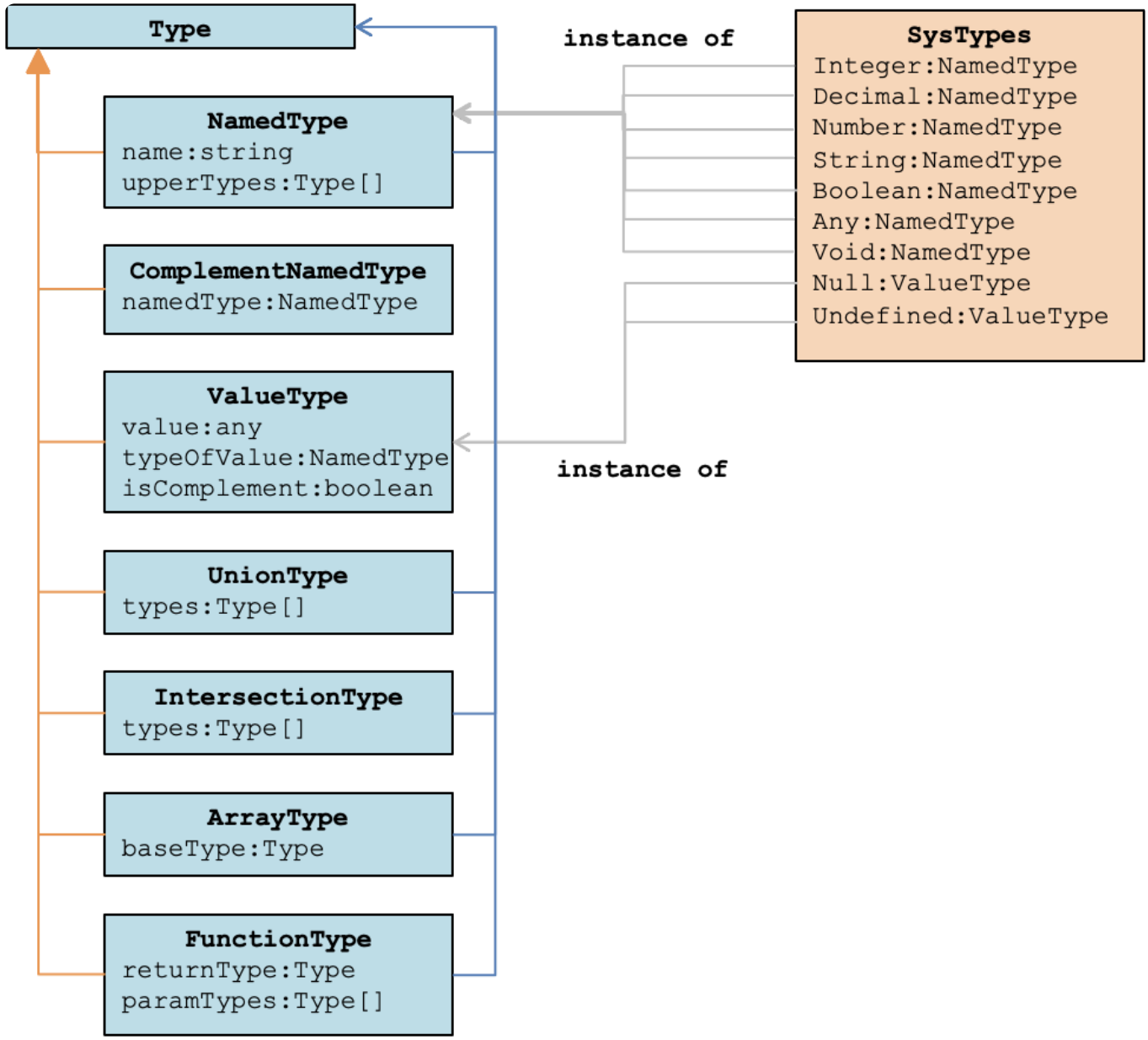
复制代码

```
let a2:number[][] = [[1,2,3],[4,5]];  //二维数组
println(a2);
println(a2[1]);             //打印其中一个维度
println(a2[0][2]);          //打印一个元素
```

它的 AST 结构如下图所示，你要注意，数组类型、数组字面量和数组元素的引用都是如何形成层层嵌套的结构的。

除了语法方面的工作以外，我们语义分析方面也要做一些增强，包括**类型处理**和**左值分析**等方面的工作。

为了实现类型处理，我们首先需要添加新的 ArrayType 类型。ArrayType 类型会引用一个基础类型。到现在为止，我们的类型体系越来越丰富了，我把原来的类型体系图更新了一版，你可以看一下：



然后，在类型处理的时候，你还需要完成这几项工作。

首先，我们要能够根据类型声明的 AST 节点，计算出变量的类型。你可以参考TypeResolver中的代码：

复制代码

```
visitArrayPrimTypeExp(te:ArrayPrimTypeExp):any{
    //求出基础类型
    let t = this.visit(te.primType) as Type;

    //创建ArrayType
    let t1 =  new ArrayType(t);

    return t1;
}
```

第二，要计算出数组字面量的类型，并检查类型是否一致。比如，下面的示例程序中，数组字面量的类型是 stirng[]，而变量 a3 的类型是 number[]，所以会检查出类型错误出来。

复制代码

```
let a3:number[] = ["Hello", "PlayScript"];
```

并且，如果数组字面量中的个别元素跟类型声明不一致，也需要能够检查出来：

复制代码

```
let a4:number[] = [2, "PlayScript"];
```

具体实现你可以参考TypeChecker中的visitArrayLiteral和visitBinary方法。代码有点多，我就不贴在文稿里了，你可以点击链接查看。

第三，我们还要能够根据下标表达式，计算每个表达式节点的类型。比如，在下面的示例程序中，a2、a2[1]、a2[0][2]的类型就需要被精确地计算出来，这样才能在编译器后端生成正确的汇编代码。

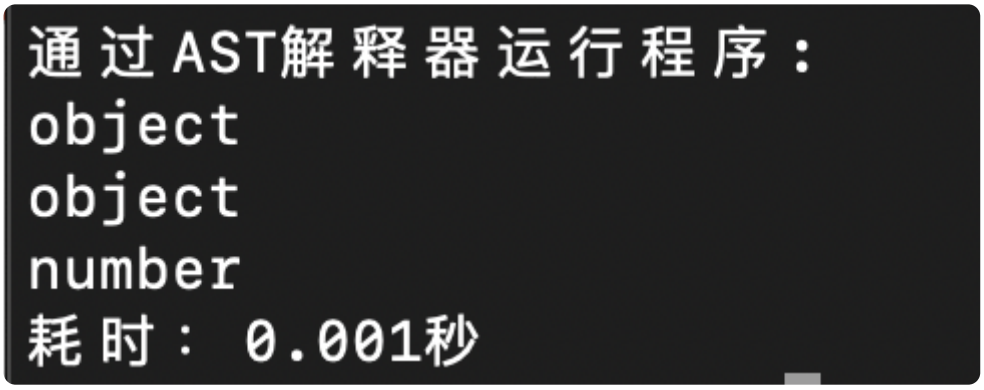
复制代码

```
let a2:number[][] = [[1,2,3],[4,5]];
println(typeof a2);         //类型是number[][]
println(typeof a2[1]);      //类型是number[]
println(typeof a2[0][2]);   //类型是number
```

你可以在语义分析后的 AST 中看到这些表达式的类型信息，我在图片中都标注出来了。

```
语义分析后的AST，注意变量和函数已被消解：
Prog
    VariableStatement
        VariableDecl a2:number[][]
            ArrayPrimType
                ArrayPrimType
                    Number
            ArrayLiteral:[[1, 2, 3], [4, 5]]:integer[][]
                ArrayLiteral:[1, 2, 3]:integer[]
                    1:integer
                    2:integer
                    3:integer
                ArrayLiteral:[4, 5]:integer[]
                    4:integer
                    5:integer
    ExpressionStatement
        FunctionCall (void)println, built-in
            typeof
                Variable: a2:number[][], resolved
    ExpressionStatement
        FunctionCall (void)println, built-in
            typeof
                IndexedExp: a2[1]:number[]
                  baseExp:
                    Variable: a2:number[][], resolved
                  subscript:
                    1:integer
    ExpressionStatement
        FunctionCall (void)println, built-in
            typeof
                IndexedExp: a2[0][2]:number
                  baseExp:
                    IndexedExp: a2[0]:number[]
                      baseExp:
                        Variable: a2:number[][], resolved
                      subscript:
                        0:integer
                  subscript:
                    2:integer
    ReturnStatement
```


你也可以看看 typeof 运算所打印出的结果。不过，TypeScript 的 typeof 运算符输出的类型信息比较有限，只能看出 a2 和 a2[1]是 object 类型，而 a2[0][2]是 number 类型，缺少更多的细节。



最后，我们还要让 ArrayType 能够像 NamedType、ValueType 等类型一样，采用集合运算的方法进行类型计算，以支持联合类型、类型窄化等场景。这里我就先不展开了，你可以看看TypeUtil中的实现。

好了，完成与类型有关的处理以后，我们还要增强一下另一个功能，就是**左值分析**。

你还记得什么是左值吗？在表达式中，能够出现在赋值运算符左边的，就叫做左值。左值相当于是一个变量的地址。你通过这个地址，可以修改变量的值。

在下面的示例程序中，我们可以给数组元素赋值，那么这些数组元素也必须是左值才可以：

复制代码

```
let a2:number[][] = [[1,2,3],[4,5]];
a2[0] = [4,5,6];
a2[0][1] = 7;
```

具体的实现，可以看一下LeftValueAttributor中的相关代码。

好了，完成上面工作以后，我们对编译器前端的增强就差不多了。接下来我们看看运行时的功能。

增强 AST 解释器

我们首先看看 AST 解释器。我们之前版本的 AST 解释器并不支持数组操作，所以我们把这个功能补上，以便跟编译成可执行程序的版本互相对比和验证。

要增强 AST 解释器，其实要做的工作还真不是太多。主要就是增加了**对下标表达式（IndexedExp）的处理**。它能返回下标表达式的值，包括左值和右值，分别用于对数组元素的写和读。

复制代码

```
visitIndexedExp(exp:IndexedExp):any{
    //下标应该是个整数
    let index = this.visit(exp.indexExp);

    let v = this.visit(exp.baseExp);
    if (exp.isLeftValue){ //返回左值
        if (v instanceof VarSymbol){
            return new ArrayElementAddress(v, [index]);
        }
        else if (v instanceof ArrayElementAddress){
            let indices = v.indices.concat([index]);
            return new ArrayElementAddress(v.varSym, indices);
        }
        else{
            console.log("Runtime error, '" + exp.baseExp.toString() + "' shoul
        }
    }
    else{ //返回右值
        if (v instanceof VarSymbol){
            let values = this.getVariableValue(v) as [];
            return values[index];
        }
        else{ //如果是多维数组，比如a[0][1]，那么访问baseExp的时候(a[0])，会返回一个一
            let values = v as[];
            return values[index];
        }
    }
}
```

这样修改完毕 AST 解释器以后，你可以用一个小小的测试程序看看它能否正常运行。比如你可以运行一下面的测试程序：

复制代码

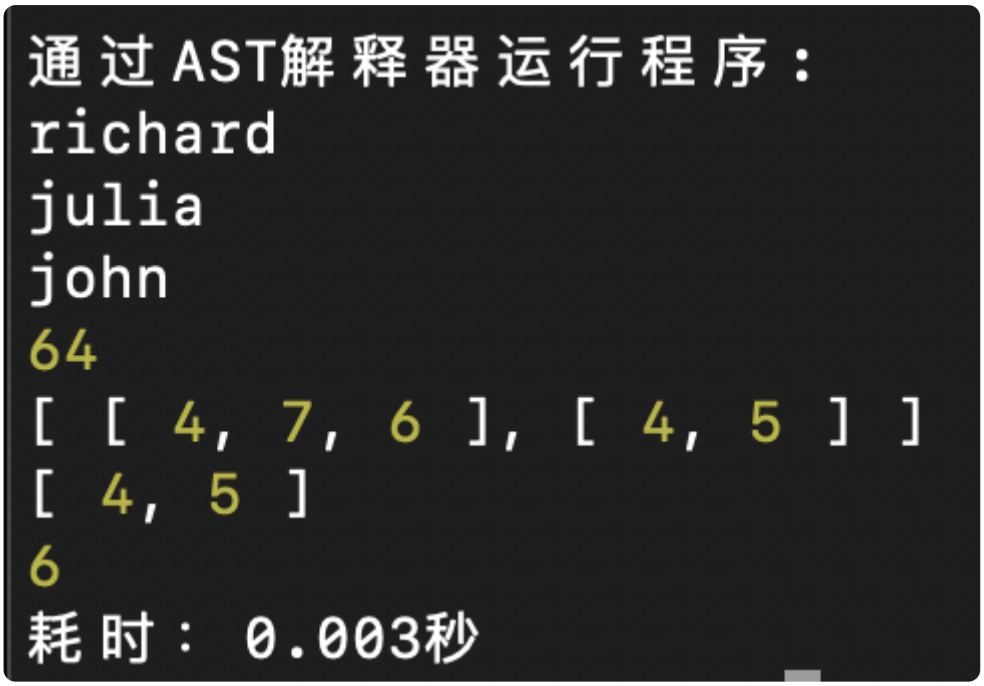
```
//字符串数组
let names:string[] = ["richard","sam", "john"];
names[1] = "julia";       //数组元素赋值，左值
for (let i = 0; i< 3; i++){
   println(names[i]);     //读取数组元素，右值
}

```

```
//number数组
let ages:number[] = [8, 18, 28];
ages[2] = 38;           //数组元素赋值，左值
let sum:number = 0;
for (let i = 0; i<3; i++){
    sum = sum + ages[i];//读取数组元素，右值
}
println(sum);

//二维数组
let a:number[][] = [[1,2,3],[4,5]];  //二维数组
a[0] = [4,5,6];          //修改一个维度
a[0][1] = 7;             //修改一个元素
println(a);              //打印二维数组
println(a[1]);           //打印一维数组
println(a[0][2]);        //打印一个元素
```

这个程序的输出结果我放在下面了，你能看到它确实能够处理数组了。



那接下来，我们再接再厉，让 PlayScript 能够把带有数组处理功能的程序编译成可执行文件。根据我们上两节课的经验，我们需要先设计一下与数组有关的内部对象以及相应的内置函数。

运行时的功能：PlayArray 对象和内置函数

我们把内部的数组对象叫做 PlayArray。PlayArray 的设计与上一节课的 PlayString 有点类似，你可以看看它的代码：

复制代码

```
typedef struct _PlayArray{
    Object object;
    //字符串的长度
    size_t length;

    //后面跟着的是数组数据
    //我们不需要保存这个指针，只需要在PlayArray对象地址的基础上增加一个偏移量就行。
    // void * data;
}PlayArray;
```

从代码中你能看出，PlayArray 对象有一个对象头、还有一个字段记录数组中的元素个数，再后面跟着的就是具体的数组数据。

并且，我们也提供了几个内置函数，来支持基本的数组操作。其中最重要的就是创建指定长度的数组。我还写了一个宏来计算数组中某个元素的地址，这样我们就能对数组进行读写操作。

复制代码

```
//创建指定元素个数的数组
PlayArray* array_create_by_length(size_t length);

//获取数组中某个元素的地址
#define PTR_ARRAY_ELEM(parr, index) (void *)(parr + sizeof(Object) + sizeof(si
```

在 PlayArray 对象的设计中，有一个问题需要你注意，就是怎么存储不同类型的数组元素。比如，PlayScript 现在已经支持了浮点型数据和字符串数据，那如何用同一个 PlayScript 对象来存储这两种数据呢？

其实，对 PlayArray 对象来说，它并不关心存储的到底是什么类型的元素，它只关心这些元素占据多少个字节的内存，便于申请内存。

对于 double 数据来说，它们占据 8 个字节 64 位。而对于 PlayString 对象来说，我们只需要存储对象的引用，也就是对象的内存地址，这个地址也是 8 个字节 64 位的。所以，现在这个版本的 PlayArray 中，我们暂且认为每个元素占据 8 个字节就好了。如果数组大小是 n，那就要为数组数据申请 n*8 个字节的内存。

你也可以看到，在上面的宏中，如果你要获取数组元素的地址，会返回一个 void *。你可以把它强制转换成其他类型，比如转换成 double *，就能存取 double 型的数据。转换成 PlayScript *，*就能存取字符串对象的引用，也就是 PlayScript*。

我用 C 语言做了一个示例程序array.c，来测试 PlayArray 对象有关的功能。你可以看一下我是如何获取数组元素的地址，并给数组元素赋值的。比如下面这个程序就是给数组元素赋 double 值：

复制代码

```
//创建示例的number数组
PlayArray* sample_array_double(){
    //创建数组
    PlayArray * parr = array_create_by_length(3);

    //给数据元素赋值
    *((double *)PTR_ARRAY_ELEM(parr,0)) = 5;
    *((double *)PTR_ARRAY_ELEM(parr,1)) = 10.5;
    *((double *)PTR_ARRAY_ELEM(parr,2)) = 10.6;

    return parr;
}
```

然后你可以再把这些数据从 double 数组里读出来，并进行汇总。

复制代码

```
//把number数组的元素汇总
double sum_array_double(PlayArray * parr){
    //读取数据并汇总
    double sum = 0;
    for (int i = 0; i< parr->length; i++){
        sum += *((double *)PTR_ARRAY_ELEM(parr,i));
    }

    return sum;
}
```

对于字符串数组，我们也可以进行类似的处理。只不过，现在存取的对象是 PlayScript*，也就是字符串引用。

复制代码

```
//创建示例的字符串数组
PlayArray* sample_array_string(){
 //创建数组
    PlayArray * parr = array_create_by_length(2);

    //给数据元素赋值
    *((PlayString **)PTR_ARRAY_ELEM(parr,0)) = string_create_by_cstr("Hello");
    *((PlayString **)PTR_ARRAY_ELEM(parr,1)) = string_create_by_cstr(" PlayScr

    return parr;
}

//把字符串数组的元素拼接在一起，形成一个新的字符串
PlayString* concat_array_string(PlayArray * parr){
    PlayString* pstr;

    if (parr->length > 0)  pstr = *((PlayString**)PTR_ARRAY_ELEM(parr, 0));

    for (int i = 1; i< parr->length; i++){
        PlayString* pstr1 = *((PlayString**)PTR_ARRAY_ELEM(parr, i));
        pstr = string_concat(pstr, pstr1);
    }

    return pstr;
}
```

甚至，你还可以往数组里存其他数组的对象引用，这也就是多维数组的情况：

复制代码

```
//创建一个示例的二维数组，其中一个维度是double类型，另一个维度是字符串数组
PlayArray * sample_array_2d(){
    //创建数组
    PlayArray * parr = array_create_by_length(2);

    *((PlayArray **)PTR_ARRAY_ELEM(parr,0)) = sample_array_double();
    *((PlayArray **)PTR_ARRAY_ELEM(parr,1)) = sample_array_string();

    return parr;
}
```

最后，你可以用 make array 命令，编译一下 array.c 示例程序，然后用./array 来运行一下这个示例程序，看看它是否成功完成了与数组有关的操作，包括创建数组、往数组里存不同类型的数据，以及从数组里读取数据。

好，现在我们已经完成了运行时的准备工作了。现在只差最后一步工作了，就是修改编译器后端，来生成正确的汇编代码，从而编译成可执行文件。

修改编译器后端

在编译器的后端，为了支持数组运算，最重要的是完成下面几项工作：**根据数组字面量来创建数组对象，获取数组元素的地址，以及对寄存器分配算法进行必要的调整。**

我们先来看看第一项任务：把数组字面量转变成数组对象。

这是在visitArrayLiteral中处理的。首先，它要调用内置函数创建一个数组对象。接着，要再计算出数组元素的地址，以便给数组的元素赋值。你可以查看一下示例代码。

第二项任务，是获取数组元素的地址。

在 array.c 示例程序中，你已经看到，在读写数组元素的时候，我们必须获取数组元素的地址。在生成汇编代码的时候，我们也要完成类似的任务。

首先，我们要根据 PlayArray 对象的地址计算出数组数据的地址。你已经知道，所有对象头的大小是 16 个字节，而记录数组元素个数的 length 字段占据 4 个字节。所以，你可以用一条指令把 PlayArray 的地址加上 20，得到数组数据的地址。

那怎么访问数组中的某个元素呢？我们在汇编代码中已经使用过内存访问的操作数，就是基于寄存器的值，再加上或者减去一个偏移量的形式。假设我们把数组数据的开头地址存放到 rax 寄存器里，那么数组的第一个元素的内存地址是 (%rax)，第二个元素的地址是 8(%rax)，第三个元素的地址是 16(%rax)，依次类推。

不过，上面这种格式是间接内存访问的简化形式。如果你还记得间接内存地址访问的完整形式，那么上述两个步骤可以合并成一条指令，比如使用 20(%rax,%rdi,8) 这样的格式可以直接计算出数组元素的地址。其中，20 是对象头的偏移量，%rdi 的值假设是 2，2 指的是第二个数组元素，8 指的是每个数组元素占据 8 个字节。所以最后的地址是：%rax + 2*8 + 20，正好就是第 2 个数组元素的地址。

第三项任务，是对寄存器分配算法进行必要的调整。

在对数组元素进行操作的时候，我们最好也先把它们挪到寄存器，在寄存器里进行相关计算。相关示例代码你可以查看 Lower 类。

在对编译器的后端进行了上述修改以后，你就可以用它来编译带有数组运算的代码，并且生成可执行程序了。你可以参考 example_array.ts示例代码，用 make example_array 命令编译成汇编代码和可执行程序。你也可以多写点不同的程序玩一玩。

看着现在我们的语言能够支持复杂的、与数组有关的语法，甚至能支持多维数组，你会不会觉得很有成就感呢？

课程小结

这就是今天的所有的内容了。今天我们用了一节课的时间，给 PlayScript 增加了处理数组的能力。我建议你记住几个关键点：

首先，是语法处理方面的知识点。在今天的语法规则中，有两处规则如果处理不好，就会变成左递归文法。比如声明多维数组类型 let a : number[][]时，以及访问多维数组元素时，如 a[i][j]。你要看看我是怎么处理这两处文法的。在之前的课程中，我们处理连续赋值表达式，比如 a=b=c 的时候，我也使用了类似的文法和解析程序。如果后面有机会，我会把这个知识点再展开给你讲一讲。

第二，是内存布局设计方面的知识点。我们用一个统一的 PlayScript 对象来保存各种数据，包括基础数据 number，以及以对象格式保存的 String，甚至其他数组对象的引用。下一节课我们会讲到自定义对象，到时候你仍然可以把这些对象的引用保存到数组里。

第三，是关于如何访问数组的元素。在语义分析阶段，我们要能够区分左值和右值的场景。左值返回的是地址，而右值返回的是地址中的值。而在编译器的后端，你需要知道如何正确的计算数组元素的地址，从而转化成正确的操作数。

思考题

目前的设计中，数组中存放的元素的大小都是 8 个字节。如果我们需要存储其他长度的元素，比如 4 个字节的整型，应该如何修改当前的设计？哪些地方的程序也需要进行调整？欢迎在留言区分享你的观点。

欢迎把这节课分享给更多感兴趣的朋友。我是宫文学，我们下节课见。

资源链接

这节课的示例代码在这里！

分享给需要的人，Ta订阅后你可得 20 元现金奖励

生成海报并分享

赞 0 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 27 | 增加更丰富的类型第2步：如何支持字符串？

精选留言

写留言

> 由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。